A Certified Procedure for RL Verification

Andrei Arusoaie, David Nowak, Vlad Rusu, Dorel Lucanu

A Certified Procedure for RL Verification

Andrei Arusoaie*, David Nowak†, Vlad Rusu‡, and Dorel Lucanu*
* Alexandru Ioan Cuza, University of Iaşi
{arusoaie.andrei, dlucanu}@info.uaic.ro
†CRIStAL, CNRS & Lille 1 University, France
david.nowak@univ-lille1.fr
‡INRIA Lille Nord-Europe, France
vlad.rusu@inria.fr

***Abstract*—Proving programs correct is hard. During the last decades computer scientists developed various logics dedicated to program verification. One such effort is Reachability Logic (RL): a language-parametric generalisation of Hoare Logic. Recently, based on RL, an automatic verification procedure was given and proved sound. In this paper we generalise this procedure and prove its soundness formally in the Coq proof assistant. For the formalisation we had to deal with all the minutiae that were neglected in the paper proof (i.e., an insufficient assumption, implicit hypotheses, and a missing case in the paper proof). The Coq formalisation provides us with a certified program-verification procedure.**

I. Introduction

Proving programs correct is one of the major challenges that computer scientists have been struggling with during the last decades. Many techniques have been developed in order to automate this non-trivial and tedious process. Nowadays there are many software tools used for proving program correctness, but proving that the verification tools themselves are correct is often avoided. In this paper we address this issue for our own work: more precisely, we present the formalisation in the Coq proof assistant of the soundness of a procedure for program verification that we proposed in [9].

In the last decades computer scientists came up with several logics meant to be used for program verification. Floyd/Hoare Logic [5], [8], Separation Logic [12], and Dynamic Logic [6] are probably the most well known logics dedicated to program verification. Recently, Matching Logic (ML) [15] and Reachability Logic (RL) [16], [17], [3] have been proposed as alternative approaches for dealing with this problem. ML and RL were inspired from the attempt to use rewrite-based operational semantics for program verification. Although operational semantics are considered too low level for verification, they are much easier to define and have the big advantage of being *executable* (and thus testable). For example, several large and complete operational semantics for real languages have been formalised with RL using the $\mathbb{K}$ framework [18], [14], [2]: C [4], [7], Java [1], JavaScript [11].

ML is a first-order logic for specifying and reasoning about program *configurations*, e.g., code and infrastructure for executing it (heap, stack, registers, etc.). Intuitively, an ML formula φ is a configuration template accompanied by a constraint, and it denotes the set of configurations that *match* the template and satisfy the constraint. RL can be used for both defining operational semantics of programming language and expressing properties about program executions. An RL formula is a pair of ML formulas, denoted $\varphi \Rightarrow \varphi'$, which says that all terminating executions that start from a configuration in the set denoted by φ, eventually reach a configuration in the set denoted by φ'. A *language-independent* and *sound* proof system, which uses the operational semantics of a language to derive RL formulas, has been proposed in [3].

One practical disadvantage of the RL proof system [3] is that the proofs tend to be very low level. Moreover, some creative user input is required when applying the rules of the proof system. Those difficulties were addressed in [9], where a procedure for program verification based on RL was proposed and the soundness of the procedure was proved. The intent behind this procedure is to overcome the lack of strategies for applying the rules of the RL proof system, and implicitly, move forward to automation. It takes as inputs sets of RL formulas representing the operational semantics of a language and the program together with its specification to be proved; if it terminates, it returns either `success` or `failure`. The soundness property states that if the procedure terminates and returns `success` then the program satisfies its specification. Its proof depends on several assumptions which are not precisely formalised in [9]. In order to provide additional confidence in this soundness result, we present in this paper a formalisation of the soundness proof in Coq.

The paper proof from [9] raised a series of technical difficulties. One difficulty is hidden in an intermediary lemma whose proof requires intricate inductive reasoning. Then, several hypotheses and assumptions are added in an ad-hoc manner just for the proofs to hold. Also, some helper lemmas are not precisely formulated, i.e., they need additional hypotheses. For a careful reader all these issues could raise some doubts about the validity of the paper proof.

A proof in a research paper should keep a good balance between correctness and clarity. The soundness proof in [9] is monolithic, i.e., the soundness does not follow from the soundness of each derivation rule, as it is commonly done in soundness proofs. Therefore, even if the procedure is recursive, the nature of the proof is not inductive; the entire execution

This work was supported by the PN III - Bridge Grant contract no. 55BG/2016: "*Creşterea imunităţii la atacuri Zero-day prin analiză statică*".

of the procedure is needed to prove that the goals are valid. This makes the proof complex and hence the task of keeping a good balance between correctness and clarity is difficult. In order to keep the proof concise, some details are omitted in [9] but this could have hidden side effects.

Clearly, a proof assistant is the solution here: it has no problem dealing with every detail because it keeps track rigorously of all cases, and thus, it ensures that no corner case is overlooked.

Contributions. The main contribution is a generalisation of the procedure and a constructive[1] formal proof in Coq of its soundness. We first generalise the procedure by introducing non-determinism and then encode it as an inductive relation in Coq. Then, we formally prove that our generalised procedure is sound. The soundness of the concrete procedure in [9] follows. Finally, we implement the procedure as a Coq function which can be used to build a certified prover.

The formalisation led us to discovering a flaw in the paper proof: it is supposed that a claim holds for a set of RL formulas, but the claim was proved only for a subset of formulas. Fortunately, this flaw does not invalidate the final result, but it makes the proof in [9] incomplete. Moreover, in order to fix the proof we realised that we need an extra hypothesis in the soundness theorem. In Coq, we add both the required hypothesis and an additional case which handles the RL formulas in question and completes the proof.

Another issue that we discovered and fixed is related to renaming the free variables in RL formulas. In [9], this is imprecisely handled using an assumption saying that certain sets of variables are disjoint, and if they are not, then the free variables can always be renamed. However, why the variable renaming is sound is not established in [9]. Here we explicitly show that free variables in RL formulas can always be renamed appropriately, and we update the proof accordingly.

During the formalisation in Coq we were able to find and fix some other imprecisions in lemmas from [9]. In order to simplify the formalisation we abstract away some details about ML that were fully stated in [9], but are in fact not relevant for soundness. Last but not least, the Coq formalisation provides us with a certified program-verification procedure. Its use in practice depends on the availability in Coq of RL-based semantics for languages; such an effort is already underway in the $\mathbb{K}$ team [10].

Related work. The closest related work is the formalisation in Coq of RL reported in [3]. Previous works [16], [17], [3] include sound and relatively complete proof systems for various versions of RL. In [3], the authors also prove the soundness of such a proof system in Coq. Our initial attempt to prove the soundness of the procedure from [9] consisted in reusing the mechanised proof reported in [3]. However, the soundness of the procedure is not a direct consequence of the soundness of the RL proof system, because there is no direct correspondence between the inference steps and the rules of the proof system. This is the reason why we choose to give a direct proof of the soundness of the procedure. An additional benefit of a direct proof is that, compared to the RL proof system, our procedure is closer to an implementation. Thus, by formalising directly the soundness of the procedure in Coq (vs. formalising the soundness of a proof system) we bridge the gap between theory and implementation. In the end we obtain a function which can be further extracted (using specialised Coq mechanisms) into a certified prover for RL.

Outline. Section II provides a short overview of ML and RL based on examples. In Section III we present the verification procedure proposed in [9], while in Section IV we describe our formalisation in Coq of the soundness of the procedure. We conclude in Section V.

II. Background

This section provides an overview of ML and RL. For both logics, we explain their syntax and semantics by means of simple examples. The precise definitions that we use in Coq are shown later in Section III. We assume the reader is familiar with (many-sorted) First-Order Logic (hereafter, FOL).

A. Matching Logic

Matching Logic is a first-order logic for specifying and reasoning about program *configurations*, e.g., code and infrastructure for executing it (heap, stack, registers, etc.).

We start diving into ML by means of a very simple example:

$$\varphi' \triangleq \langle \texttt{x = x + y; y = x - y; x = x - y;} \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y \rangle \wedge x{+}y < 2^{32}$$

For someone familiar with FOL, the formula above may look strange: what is the content surrounded by '$\langle$' and '$\rangle$' and why is it used in the left hand side of a conjunction? Recall that ML is designed to specify and reason about program configurations. In ML, the construct $\langle \texttt{x = x + y; y = x - y; x = x - y;} \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y \rangle$ is a formula and represents all the configurations that *match* it. For this particular example, the configurations matched by the formula consist of a *program* (i.e., the sequence of assignments which is supposed to swap two values) and a *state* containing mappings from program variables to their values (here, x and y are variables that can be matched by values). For real-life languages (like C [4], [7] or Java [1]) configurations typically hold more information.

The formal syntax of ML formulas is shown below. Let Σ be a many-sorted algebraic signature, Π a set of predicate symbols, Var a set of (sorted) variables, and *Cfg* a *sort* for program configurations; then, the syntax of ML formulas is:

$$\varphi ::= \pi \mid \top \mid p(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid (\exists V)\varphi,$$

where π is a Σ-term of sort *Cfg* with variables (also called *basic pattern*)[2], p is a predicate symbol in Π, t_i are Σ-terms with variables of appropriate sorts, and V a subset of Var. As usual, the other known FOL connectives ($\vee$, $\rightarrow$, ...) and quantifier $\forall$ can be expressed using the ones above.

Example 1: Recall the ML formula φ' shown at the beginning of this section. Let $\pi \triangleq \langle \texttt{x = x + y; y = x - y; x = x - y;} \mid$

[1]We thank the reviewers for pointing it out.

[2]Here we only consider the "topmost version" of ML from [13], that is, π is a term of sort *Cfg* with variables.

$\texttt{x} \mapsto x\ \texttt{y} \mapsto y\rangle$. Then π is a term of sort *Cfg* with variables x and y. The constructor for such terms is $\langle \_ \mid \_ \rangle$, where the two occurrences of _ are placeholders for terms of a sort *Pgm* (corresponding to programs) and terms of a sort *Map* (representing the state). Note that x and y are different from x and y: the former are logical variables included in *Var* while the latter are constants representing program variables (or identifiers) of sort *Id*. Symbols + and $+$ that occur in $\pi \quad \wedge \quad x + y \ < \ 2^{32}$ are also different: the first is an operation symbol over program expressions, while the latter is the integer addition that applies only to integers. Here we distinguish them by different font sizes (+ vs. $+$). All these operation symbols and their corresponding sorts are included in Σ, while the predicates (e.g., $<$) are in Π.

For ML formulas we use the same notion of *free variable* as in FOL, which we denote by $\mathrm{FV}(\varphi)$ in the paper. The existential closure of φ is denoted by $\overline{\varphi} \triangleq (\exists\ \mathrm{FV}(\varphi))\varphi$.

Next, let us clarify how ML formulas are interpreted. Intuitively, an ML formula is a pattern which denotes the set of concrete configurations that *match* that pattern.

Example 2: The following term:

$$\gamma \triangleq \langle \texttt{x = x + y; y = x - y; x = x - y; } \mid \texttt{x} \mapsto 5\ \texttt{y} \mapsto 10\rangle$$

is a configuration that matches the ML formula φ': the structural part of γ is the same as the one of φ' where variables x and y are replaced by concrete values 5 and 10; also, for $x = 5$ and $y = 10$ the inequality $x + y < 2^{32}$ holds.

Formally, the satisfaction relation $\models_{\text{ML}}$ of ML is akin to the satisfaction relation of FOL (denoted $\models_{\text{FOL}}$). Next, we consider a (Σ, Π)-model M. Then, $(\gamma, \rho) \models_{\text{ML}} (\exists V)\varphi$ iff there is $\rho' : \mathit{Var} \to M$ with $\rho'(x) = \rho(x)$ for all $x \notin V$ such that $(\gamma, \rho') \models_{\text{ML}} \varphi$. This is similar to FOL, except that $\models_{\text{ML}}$ is defined over pairs (γ, ρ), where γ is an element in M of sort *Cfg* called a concrete *configuration*, $\rho : \mathit{Var} \to M$ is a valuation, and ML formulas φ. In addition to the other FOL constructs, $\models_{\text{ML}}$ includes the ML particular case:

$$(\gamma, \rho) \models_{\text{ML}} \pi \text{ iff } \rho(\pi) = \gamma.$$

Example 3: As already pointed out in Example 2, γ is a configuration that matches φ'. Formally, if $\rho : \mathit{Var} \to M$ is a valuation such that $\rho(x) = 5$ and $\rho(y) = 10$, then $(\gamma, \rho) \models_{\text{ML}} \varphi'$ holds: first[3], $\rho(\langle \texttt{x = x + y; y = x - y; x = x - y; } \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y\rangle) = \gamma$, and second, $\rho \models_{\text{FOL}} x + y < 2^{32}$.

It has been shown in [16] that ML can be encoded in FOL. We discuss here a variant of this encoding. If φ is an ML-formula then its encoding $\varphi^{=?}$ in FOL is the formula $(\exists z)\varphi'$, where φ' is obtained from φ by replacing each basic pattern occurrence π with $z = \pi$, and z is a variable which does not appear in the free variables of φ. Here are a few examples of formulas and their encodings:

φ	$\varphi^{=?}$
$(\pi_1 \wedge \phi_1) \vee (\pi_2 \wedge \phi_2)$	$(\exists z)((z = \pi_1 \wedge \phi_1) \vee (z = \pi_2 \wedge \phi_2))$
$\pi_1 \wedge \neg\pi_2$	$(\exists z)((z = \pi_1) \wedge \neg(z = \pi_2))$

The encoding of a formula φ has the following property [9]: *for all ρ and φ, $\rho \models_{\text{FOL}} \varphi^{=?}$ iff exists γ such that $(\gamma, \rho) \models_{\text{ML}} \varphi$.*

[3]For simplicity, we apply the valution ρ directly to a *Cfg* term; instead, we should have used the homomorphic extension of ρ to terms of sort *Cfg*.

B. Reachability Logic

We have seen in the previous section that ML formulas denote sets of program configurations (or states). An RL formula (also called *rule*) is a pair of ML formulas, written $\varphi \Rightarrow \varphi'$, that expresses reachability relationships between the two sets of configurations denoted by φ and φ'. For example:

$$\langle \texttt{x = x + y; y = x - y; x = x - y; } \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y\rangle \wedge x + y < 2^{32}$$
$$\Rightarrow$$
$$\langle \texttt{skip} \mid \texttt{x} \mapsto y\ \texttt{y} \mapsto x\rangle$$

is an RL formula which expresses the fact that configurations which satisfy[4] $\langle \texttt{skip} \mid \texttt{x} \mapsto y\ \texttt{y} \mapsto x\rangle$ will be reached from configurations which satisfy $\langle \texttt{x = x + y; y = x - y; x = x - y; } \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y\rangle \wedge x + y < 2^{32}$. Informally, the formula expresses the fact that if we execute the assignments, then the values of program variables x and y are swapped. For this particular program, this is a desired property to prove. It is important to note how expressive RL is: in the left hand side of the RL formula we included the program and a side condition (which ensures that integer overflow does not happen for 32 bits architectures); the right hand side expresses that no code is left to be executed, and variables x and y hold the swapped initial values.

We extend the FV construct over RL formulas too; by $\mathrm{FV}(\varphi \Rightarrow \varphi')$ we denote the union $\mathrm{FV}(\varphi) \cup \mathrm{FV}(\varphi')$. Also, if $\mathcal{S}$ is a set of RL formulas then $\mathrm{FV}(\mathcal{S}) \triangleq \bigcup_{\varphi \Rightarrow \varphi' \in \mathcal{S}} \mathrm{FV}(\varphi \Rightarrow \varphi')$.

A set of RL formulas $\mathcal{S}$ defines a transition system, denoted $(\mathit{Cfg}, \Rightarrow_{\mathcal{S}})$, over configurations: we say that $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1$ if there is a rule $\varphi_0 \Rightarrow \varphi_1 \in \mathcal{S}$ and there is a valuation $\rho' : \mathit{Var} \to M$ such that $(\gamma_0, \rho') \models_{\text{ML}} \varphi_0$ and $(\gamma_1, \rho') \models_{\text{ML}} \varphi_1$. We often use only $\Rightarrow_{\mathcal{S}}$ to denote the transition system $(\mathit{Cfg}, \Rightarrow_{\mathcal{S}})$.

Example 4: The set $\mathcal{S} \triangleq \{\alpha^+, \alpha^-, \alpha\}$ of RL formulas defines a transition system:

$$\alpha^+ \triangleq \langle X{=}Z_1 + Z_2; \mathit{Code} \mid Z_1 \mapsto V_1\ Z_2 \mapsto V_2\rangle \Rightarrow \langle X{=}V_1 + V_2; \mathit{Code} \mid Z_1 \mapsto V_1\ Z_2 \mapsto V_2\rangle$$

$$\alpha^- \triangleq \langle X{=}Z_1 - Z_2; \mathit{Code} \mid Z_1 \mapsto V_1\ Z_2 \mapsto V_2\rangle \Rightarrow \langle X{=}V_1 - V_2; \mathit{Code} \mid Z_1 \mapsto V_1\ Z_2 \mapsto V_2\rangle$$

$$\alpha \triangleq \langle X{=}V; \mathit{Code} \mid (X \mapsto V')\ \mathit{State}\rangle \Rightarrow \langle \mathit{Code} \mid X \mapsto V\ \mathit{State}\rangle$$

Here, X, Z_1 and Z_2 are variables of sort *Id*; V, V', V_1, V_2 are variables of sort *Int* which are supposed to match over integer values; Code is a variable of sort *Pgm* and State is a variable of sort *Map*. Also, recall that + (−) and $+$ (respectively, $-$) are different symbols: the first one is used in the language expressions, while the second is the integer addition (subtraction). The first two rules are meant to capture the evaluation of addition and subtraction of expressions consisting only of program variables. Essentially, the first rule expresses the fact that if we have to evaluate the program expression Z_1 + Z_2, where Z_1 and Z_2 are variables of sort *Id*, then we replace it with the integer expression $V_1 + V_2$, where

[4]It refers to the ML satisfaction relation.

V_1 and V_2 hold the values of the program variables matched by Z_1 and Z_2. The rule α^- does the same thing for subtraction.

Finally, α corresponds to the step-by-step execution of assignments: if $X{=}V; Code$ needs to be executed then the current value of the program variable matched by X in the state, namely V', is replaced (in the state in the right hand side) by the new integer value V. Also, in the right hand side $Code$ holds the remaining assignments to be executed. Note that the rest of the state matched by $State$ remains unchanged. The set $\mathcal{S}$ can be regarded as an operational semantics.

The satisfaction relation $\models_{\text{RL}}$ of RL is defined using *paths*, which are (possibly infinite) sequences of transitions of the form $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \gamma_2 \Rightarrow_{\mathcal{S}} \cdots$.

Example 5: Recall the set $\mathcal{S} \triangleq \{\alpha^+, \alpha^-, \alpha\}$ from Example 4. The following sequence represents a path:

$$\begin{aligned}
\tau \triangleq\ & \langle \texttt{x = x + y; y = x - y; x = x - y;} \mid \texttt{x} \mapsto 5\ \texttt{y} \mapsto 10\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{x =5 + 10; y = x - y; x = x - y;} \mid \texttt{x} \mapsto 5\ \texttt{y} \mapsto 10\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{y = x - y; x = x - y;} \mid \texttt{x} \mapsto 15\ \texttt{y} \mapsto 10\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{y =15 - 10; x = x - y;} \mid \texttt{x} \mapsto 15\ \texttt{y} \mapsto 10\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{x = x - y;} \mid \texttt{x} \mapsto 15\ \texttt{y} \mapsto 5\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{x =15 - 5;} \mid \texttt{x} \mapsto 15\ \texttt{y} \mapsto 5\rangle \Rightarrow_{\mathcal{S}} \\
& \langle \texttt{skip} \mid \texttt{x} \mapsto 10\ \texttt{y} \mapsto 5\rangle
\end{aligned}$$

Here we considered that `skip` is the empty sequence; also, we evaluated the integer expressions in the state. The rules applied, in order, are: α^+, α, α^-, α, α^-, α.

If none of the rules in $\mathcal{S}$ can be applied to a configuration then we say that the configuration is *terminating*. For instance, the last configuration of the path shown in Example 5 is terminating. Paths can be infinite too. A typical example where infinite paths can occur are the rules for handling loops in programs. A path τ is *complete* if it is either finite and the last configuration in τ is terminating, or it is infinite. Since the path shown in Example 5 is finite and terminating then it is also complete. Moreover, we say that a path τ *starts from* from an ML formula φ if the first configuration in the path is matched by φ. The path shown in Example 5 starts from:

$\varphi' \triangleq \langle \texttt{x = x + y; y = x - y; x = x - y;} \mid \texttt{x} \mapsto x\ \texttt{y} \mapsto y\rangle \wedge x{+}y < 2^{32}$

In this paper we are only interested in the *all-path* interpretation [3] of RL formulas: a pair (τ, ρ) *satisfies* an RL formula $\varphi \Rightarrow \varphi'$, written $(\tau, \rho) \models \varphi \Rightarrow \varphi'$, iff (τ, ρ) starts from φ (recall that $\tau \triangleq \gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \gamma_2 \Rightarrow_{\mathcal{S}} \cdots$), and either there exists $i \geq 0$ such that $(\gamma_i, \rho) \models_{\text{ML}} \varphi'$, or τ is infinite.

Example 6: Consider τ the path shown in Example 5 and φ' the ML formula shown above. Then, $(\tau, \rho) \models \varphi' \Rightarrow \langle \texttt{skip} \mid \texttt{x} \mapsto y\ \texttt{y} \mapsto x\rangle$, with valuation ρ chosen such that $\rho(x) = 5$ and $\rho(y) = 10$. Clearly, γ_0 (i.e., the first configuration of τ) is an instance of φ' (as shown in Example 2). Also, there is an $i = 6$, such that $\gamma_i \triangleq \langle \texttt{skip} \mid \texttt{x} \mapsto 10\ \texttt{y} \mapsto 5\rangle$ is matched by $\langle \texttt{skip} \mid \texttt{x} \mapsto y\ \texttt{y} \mapsto x\rangle$, using the same ρ.

Finally, we say that $\Rightarrow_{\mathcal{S}}$ *satisfies* $\varphi \Rightarrow \varphi'$, written $\Rightarrow_{\mathcal{S}} \models_{\text{RL}} \varphi \Rightarrow \varphi'$, iff $(\tau, \rho) \models_{\text{RL}} \varphi \Rightarrow \varphi'$ for all (τ, ρ) starting from φ with τ complete. We often write $\mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$ instead of $\Rightarrow_{\mathcal{S}} \models_{\text{RL}} \varphi \Rightarrow \varphi'$.

C. Derivatives of an ML (and RL) formula

A notion introduced in [9] is that of *derivative* of an ML (and RL) formula. If φ is an ML formula and $\mathcal{S}$ is a set of RL formulas, then the derivative of φ with $\mathcal{S}$ is defined as follows:

$\Delta_{\mathcal{S}}(\varphi) \triangleq \{(\exists \text{FV}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \mid \varphi_l \Rightarrow \varphi_r \in \mathcal{S}\}$.

If $\varphi \Rightarrow \varphi'$ is an RL formula then

$$\Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi') \triangleq \{\varphi_1 \Rightarrow \varphi' \mid \varphi_1 \in \Delta_{\mathcal{S}}(\varphi)\}.$$

Intuitively, the derivative of an ML formula φ encodes the *concrete successors* by $\Rightarrow_{\mathcal{S}}$ of configurations matching φ. Derivatives can be computed for sets of RL formulas G too:

$$\Delta_{\mathcal{S}}(G) \triangleq \bigcup_{\varphi \Rightarrow \varphi' \in G} \Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi').$$

Example 7: We extend the set $\mathcal{S} \triangleq \{\alpha^+, \alpha^-, \alpha\}$ shown in Example 4 with the following rules:

$\alpha^t \triangleq \langle \texttt{while}\ (R{>=}B)\{S\}; Code \mid State\ B{\mapsto}b\ R{\mapsto}r\rangle \wedge r \geq b \Rightarrow \langle S; \texttt{while}\ (R{>}B)\{S\}; Code \mid State\ B \mapsto b\ R \mapsto r\rangle$

$\alpha^f \triangleq \langle \texttt{while}\ (R{>=}B)\{S\}; Code \mid State\ B{\mapsto}b\ R{\mapsto}r\rangle \wedge r < b \Rightarrow \langle Code \mid State\ B \mapsto b\ R \mapsto r\rangle$

These rules simulate the behavior of a very particular `while` loop, where conditions are of the form $R >= B$. The rule α^t essentially says that if the values r and b corresponding to variables R and B satisfy $r \geq b$, then we execute the loop body S once, and then again the loop. On the other hand, when $r < b$, the loop is not executed and the execution continues with $Code$ as stated by α^f. We consider the following patterns:

$\varphi \triangleq \langle \texttt{while}\ (\texttt{r>=b})\{\texttt{d=d+1; r=r-b}\} \mid \texttt{a}{\mapsto}a\ \texttt{b}{\mapsto}b\ \texttt{d}{\mapsto}d\ \texttt{r}{\mapsto}r\rangle \wedge a = b*d+r \wedge a \geq 0 \wedge b > 0$

$\varphi' \triangleq \langle \texttt{skip} \mid \texttt{a} \mapsto a\ \texttt{b} \mapsto b\ \texttt{d} \mapsto a/b\ \texttt{r} \mapsto a\%b\rangle$

Here $/$ and $\%$ represent the integer division and the remainder operations. Let $G_0 \triangleq \{\varphi \Rightarrow \varphi'\}$. The set of derivatives computed for G_0 is $\Delta_{\mathcal{S}}(G_0) = \{\varphi_1 \Rightarrow \varphi', \varphi_2 \Rightarrow \varphi'\}$, where:

$\varphi_1 \triangleq \langle \texttt{d=d+1; r=r-b; while}\ (\texttt{r>=d})\{\texttt{d=d+1; r=r-b}\} \mid \texttt{a}{\mapsto}a\ \texttt{b}{\mapsto}b\ \texttt{d}{\mapsto}d\ \texttt{r}{\mapsto}r\rangle \wedge a = b{*}d{+}r \wedge r \geq b \wedge a \geq 0 \wedge b > 0$

$\varphi_2 \triangleq \langle \texttt{skip} \mid \texttt{a}{\mapsto}a\ \texttt{b}{\mapsto}b\ \texttt{d}{\mapsto}d\ \texttt{r}{\mapsto}a\rangle \wedge a = b{*}d{+}r \wedge r < b \wedge a \geq 0 \wedge b > 0$

Both φ_1 and φ_2 are in fact the *symbolic successors* of φ using $\mathcal{S}$. Moreover, the configurations that match φ_1 and φ_2 are the concrete successors by $\Rightarrow_{\mathcal{S}}$ of configurations matching φ.

III. A PROCEDURE FOR VERIFYING RL PROPERTIES

A procedure for verifying RL formulas (Figure 1), called `prove`, was introduced in [9], and is meant to overcome the lack of strategies for applying the rules of the RL proof system from [3], and implicitly, to move forward towards automation. As the RL proof system, the procedure has a coinductive nature. Intuitively, given an RL formula $\varphi \Rightarrow \varphi'$, one can symbolically execute φ and check whether every leaf node of the obtained symbolic execution tree implies φ'. The procedure uses derivatives to compute symbolic execution paths. An obvious problem of this approach is that the symbolic execution tree can be infinite due to loops or recursion. To overcome this problem, RL allows the use of helper formulas called *circularities*, which generalise the notion of invariant.

Instead of proving a single formula, `prove` attempts to prove a set of formulas (goals). The procedure checks first whether a helper formula can be used (e.g., an RL formula specifying a loop) to discharge the current goal, rather than performing symbolic execution blindly. A goal can be used in the proof of a different goal or in its own proof, provided that at least one symbolic step has been performed from it.

The procedure takes as input three sets of RL formulas: the operational semantics $\mathcal{S}$, a set of (initial) goals G_0, and a recursive argument G whose initial value is $\Delta_{\mathcal{S}}(G_0)$; and it returns either `success` or `failure`. At each recursive call, a goal from G is either eliminated and/or replaced by other goals. If $\varphi \Rightarrow \varphi' \in G$ is the current goal then it is processed as follows: if $M \models_{\text{ML}} \varphi \rightarrow \varphi'$ then the goal is eliminated from G; else, the procedure checks whether there is a circularity available in G_0 which is used to generate a new goal that replaces the current one; if no circularity is found, but φ is $\mathcal{S}$-derivable, then $\varphi \Rightarrow \varphi'$ is replaced by a set of goals consisting in its symbolic successors; otherwise, `prove` returns `failure`. The procedure succeeds when all the goals in $\Delta_{\mathcal{S}}(G_0)$ together with those generated during the execution are eliminated. If the procedure does not terminate or it returns `failure` then it means that G_0 does not contain enough information to prove the goals. This problem is similar to finding appropriate invariants for loops in imperative programs.

Example 8: Recall the rules $\mathcal{S} = \{\alpha^+, \alpha^-, \alpha, \alpha^t, \alpha^f\}$ from Examples 7 and 4. Also, recall the set $G_0 = \{\varphi \Rightarrow \varphi'\}$ and patterns φ_1 and φ_2 from Example 7. We pass to `prove` the sets $\mathcal{S}$, G_0, and $G = \Delta_{\mathcal{S}}(G_0) = \{\varphi_1 \Rightarrow \varphi', \varphi_1 \Rightarrow \varphi'\}$. The procedure picks a goal to be proved from G, say $\varphi_2 \Rightarrow \varphi'$ and checks whether $M \models_{\text{ML}} \varphi_2 \rightarrow \varphi'$. In this case, the implication holds, since $a = b * d + r \wedge r < b \wedge a \geq 0 \wedge b > 0$ implies $a/b = d$ and $r = a\%b$; thus, the values of `d` and `r` in the state are correct. This goal is now eliminated from G.

Next, the only goal left to prove is $\varphi_1 \Rightarrow \varphi'$. Since the implication $M \models_{\text{ML}} \varphi_1 \rightarrow \varphi'$ does not hold, and no circularities $\varphi_c \Rightarrow \varphi'_c$ are available in G_0 such that $M \models_{\text{ML}} \varphi_1 \rightarrow \overline{\varphi}_c$, then `prove` computes the next successor using α^+:

$$\varphi'_1 \triangleq \langle \texttt{r=r-b;while (r>=d)\{d=d+1;r=r-b\}} \mid \texttt{a}\mapsto a\ \texttt{b}\mapsto b\ \texttt{d}\mapsto d+1\ \texttt{r}\mapsto r\rangle \wedge a = b*d+r \wedge r \geq b \wedge a \geq 0 \wedge b > 0$$

The same case for $\varphi'_1 \Rightarrow \varphi'$: the implication and circularity tests fail, and α^- is used to get the next successor:

$$\varphi''_1 \triangleq \langle \texttt{while (r>=d)\{d=d+1;r=r-b\}} \mid \texttt{a}\mapsto a\ \texttt{b}\mapsto b\ \texttt{d}\mapsto d+1\ \texttt{r}\mapsto r-b\rangle \wedge a = b*d+r \wedge r \geq b \wedge a \geq 0 \wedge b > 0$$

Now, `prove` reaches a point where the circularity $\varphi \Rightarrow \varphi'$ can be applied, since $M \models_{\text{ML}} \varphi''_1 \rightarrow \overline{\varphi}$. This is because $a = b*d+r \wedge r \geq b \wedge a \geq 0 \wedge b > 0$ implies $a = b*(d+1)+(r-b) \wedge a \geq 0 \wedge b > 0$. Therefore, the circularity is applied and `prove` eliminates the last goal from G' and returns `success`.

IV. A FORMAL PROOF OF SOUNDNESS IN COQ

In this section we generalise the procedure (Figure 1) by introducing non-determinism and then encode it as an inductive relation in Coq. In spite of this generalisation, the formal proof of soundness follows the same pattern as the paper proof in [9]. Using Coq, we were able to find a flaw in the paper proof: a claim about the goals in G_0 does not hold. This flaw does not invalidate the final result, but it makes the proof incomplete: an additional case is needed to prove the conclusion of an important lemma for the goals in G_0. Note that, in the proof of the soundness theorem, the goals (of interest) for which we apply that lemma are exactly those from G_0. During the formalisation, we also realised that we need an extra hypothesis in the soundness theorem to fix the proof. All these issues are fixed in the Coq formalisation.

procedure `prove`$(\mathcal{S}, G_0, G)$

1: **if** $G = \emptyset$ **then return** `success`
2: **else choose** $\varphi \Rightarrow \varphi' \in G$
3: **if** $M \models_{\text{ML}} \varphi \rightarrow \varphi'$ **then return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\})$
4: **else if** there is $\varphi_c \Rightarrow \varphi'_c \in G_0$ s. t. $M \models_{\text{ML}} \varphi \rightarrow \overline{\varphi}_c$ **then**
5: **return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \Delta_{\varphi_c \Rightarrow \varphi'_c}(\varphi \Rightarrow \varphi'))$
6: **else if** φ is $\mathcal{S}$-derivable **then**
7: **return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi'))$
8: **else return** `failure`.

Fig. 1. RL verification procedure. $\overline{\varphi}_c$ denotes $(\exists \text{FV}(\varphi_c))\varphi_c$.

Unlike in [9], in the Coq proof we explicitly handle the renaming of variables in RL formulas. Also, we reformulate several lemmas from [9] (Lemmas 5, 7, and 8) such that they are self contained, and we update their proofs accordingly.

The theoretical results shown in this section are fully formalized in the Coq proof assistant.

A. ML *in Coq*

A definition of ML such as in [9] would require us to formalise many things (algebraic specifications, FOL, etc.). All these details are in fact a complete language definition, i.e., a triple $((\Sigma, \Pi, \mathit{Cfg}), M, \mathcal{S})$. However, for the soundness of the verification procedure all these details are irrelevant. Moreover, we want our procedure to remain language parametric. For this, we introduce a set of axioms that capture only what is needed in the soundness proof: variables, conjunction($\wedge$), implication ($\rightarrow$), the existential quantifier ($\exists$), a model M together with valuations, a set of rules $\mathcal{S}$, and the ML satisfaction relation. When all these axioms are instantiated (i.e., a particular language is provided), the procedure remains sound.

Variables (*Var*), program configurations (*State*), models (*Model*), and valuations (*Valuation*) are all parameters in Coq. From the ML syntax we only keep $\rightarrow$, $\wedge$, and $\exists$, together with the corresponding axioms of the ML satisfaction relation $\models_{\text{ML}}$. To avoid ambiguities, we distinguish the logical connectives and quantifiers of ML ($\wedge$, $\rightarrow$, $\exists$, ...) from those of Coq by using bold font for the latter ($\mathbf{\wedge}$, $\mathbf{\rightarrow}$, $\mathbf{\exists}$, ...).The validity of an ML formula φ in model M is denoted by $\models^{M}_{\text{ML}}$[5].

To collect the free variables of an ML formula we use a function called FV. In Coq, FV is a parameter and needs an implementation when the ML constructs are instantiated.

In the proofs of our lemmas we often have to build new valuations from existing ones. For this we use a function ϱ' which exhibits the following behaviour: $\varrho(\rho, \rho', x)$ returns a new valuation which applied to variable x produces the same

[5]These details are shown in the Appendix.

result as valuation ρ' applied to x; for all the other variables $z(\neq x)$ the valuation $\varrho(\rho, \rho', z)$ is equal to ρ. We also use the function ϱ which is an extension of ϱ' to sets of variables. The corresponding Coq definitions and lemmas (which are proved in Coq) are shown below:

```
Definition ϱ' (ρ ρ':Valuation)(x : Var) : Valuation :=
        fun z => if (x = z) then ρ'(x) else ρ(z).

Fixpoint ϱ (ρ ρ':Valuation)(V : list Var) : Valuation :=
              match V with
                    | [] => ρ
                    | v :: vs => ϱ'(ϱ(ρ, ρ', v), ρ', V)
              end.
Lemma ϱ^∉ :   ∀x ρ ρ' V . x ∉ V → ϱ(ρ, ρ', V)(x) = ρ(x)
Lemma ϱ^∈ :   ∀x ρ ρ' V . x ∈ V → ϱ(ρ, ρ', V)(x) = ρ'(x)
```

If an ML formula φ is satisfied by a pair (γ, ρ), and none of its free variables are in V, then for any ρ', the pair $(\gamma, \varrho(\rho, \rho', V))$ also satisfies φ, because on those variables $\varrho(\rho, \rho', V)$ has the same effect as ρ. Conversely, if φ is satisfied by a pair (γ, ρ'), and all its free variables are included in V, then for any ρ, the pair $(\gamma, \varrho(\rho, \rho', V))$ also satisfies φ. Since the full definition of $\models_{\text{ML}}$ is abstract here, these properties are given as axioms:

```
Axiom ⊨^∉_ϱ : ∀ φ γ ρ ρ' V. (∀x . x ∈ FV(φ) → x ∉ V)∧
                              (γ, ρ) ⊨ML φ → (γ, ϱ(ρ, ρ', V)) ⊨ML φ

Axiom ⊨^⊆_ϱ : ∀ φ γ ρ ρ' V. FV(φ) ⊆ V∧
                              (γ, ρ') ⊨ML φ → (γ, ϱ(ρ, ρ', V)) ⊨ML φ
```

Another property which depends on $\models_{\text{ML}}$ is the encoding of ML formulas into FOL: *for all ρ and φ, $\rho \models_{\text{FOL}} \varphi^{=?}$ iff there is γ such that $(\gamma, \rho) \models_{\text{ML}} \varphi$*. Here we do not define FOL, but we use the fact that any FOL formula is also an ML formula. Hence, the encoding will transform an ordinary ML formula into an ML formula without patterns (i.e., a FOL formula), and the property above is axiomatised as:

```
Parameter φ^=? : MLFormula → MLFormula
Axiom ⊨^=? : ∀ γ' ρ φ. (γ', ρ) ⊨ML φ^=? ↔ (∃γ)(γ, ρ) ⊨ML φ
```

B. RL *and derivatives*

By definition, an RL formula is just a pair of ML formulas:

```
Definition RLFormula := (MLFormula * MLFormula)
```

For simplicity, in Coq we use the notation $\varphi \Rightarrow \varphi'$ instead of the pair notation (φ, φ').

In [9] the following assumption is made: for any RL formula $\varphi \Rightarrow \varphi'$ the condition $\text{FV}(\varphi') \subseteq \text{FV}(\varphi)$ holds. In Coq we introduce the notion of *well-formed* RL formula:

```
Definition wf(φ ⇒ φ') := FV(φ') ⊆ FV(φ)
```

As shown in Section II-B, a set of RL formulas $\mathcal{S}$ defines a transition system over states. We assume a given set of RL formulas $\mathcal{S}$ and we define the transition relation $\Rightarrow_{\mathcal{S}}$ as:

```
Variable S : list RLFormula
Definition γ ⇒_S γ' := ∃φ φ' ρ . φ ⇒ φ' ∈ S ∧
                              (γ, ρ) ⊨ML φ ∧ (γ', ρ) ⊨ML φ'
Definition total(S) := ∀γ ρ φ . (γ, ρ) ⊨ML φ → ∃γ'.γ ⇒_S γ'
```

The satisfaction relation $\models_{\text{RL}}$ of RL is defined over execution paths τ, which could be either finite or infinite. Because of that, we formalise them as functions from *nat* to `option` *State* (where `option` *State* is the extension of *State* with an extra element `None`). The i^{th} element of a path τ is $\tau(i)$ and can be either a state γ_i or `None`. The subpath of τ which starts at position i is denoted $\tau|_i$. Also, a path is *well-formed*, written *wfPath*(τ), if every two consecutive states (say γ_i and γ_{i+1}) are in the transition relation given by $\mathcal{S}$ ($\gamma_i \Rightarrow_{\mathcal{S}} \gamma_{i+1}$), and for all i and j such that $i < j$, if $\tau(i) = \texttt{None}$ then $\tau(j) = \texttt{None}$.

A path τ is infinite, written *infinite*(τ), if for all i, $\tau(i) \neq \texttt{None}$. A configuration γ is terminating, denoted as *terminating*(γ), if there is no γ' such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. A well-formed path τ is complete and has n transitions, written *complete*(τ, n) if τ is finite and *terminating*$(\tau(n))$. Also, a well-formed path τ is complete, denoted by *complete*(τ), if *infinite*(τ) or *complete*(τ, n) for some natural number n. Note that, if the path τ is complete and well-formed then for all i (if τ is finite then $i \leq n$) the subpath $\tau|_i$ is also complete and well-formed. This statement is assumed in [9] but in Coq we prove it explicitly. Finally, we say that (τ, ρ) *startsFrom* φ if $(\tau(0), \rho) \models_{\text{ML}} \varphi$. All these are defined in Coq as follows:

```
Definition Path := nat → option State
Definition wfPath(τ) := ∀ i j.(i<j→τ(i)=None→τ(j)=None)∧
                        ∀ i. (τ(i) ≠ None ∧ τ(i + 1) ≠ None)
                             → ∃ γ γ'. (τ(i) = γ ∧ τ(i + 1) = γ' ∧ γ ⇒_S γ')

Definition infinite(τ) := wfPath(τ) ∧ (∀i)τ(i) ≠ None
Definition hasLen(τ, n) := wfPath(τ) ∧ ¬infinite(τ)∧
                           ∃ n γ. τ(n) = γ ∧ τ(n + 1)=None

Definition terminating(γ) := (∀γ')¬(γ ⇒_S γ')
Definition complete(τ, n) := infinite(τ) ∨ (hasLen(τ, n)∧
                                 terminating(τ(n)))
```

The satisfaction relation of RL is defined for complete paths τ: $(\tau, \rho) \models_{\text{RL}} \varphi \Rightarrow \varphi'$ iff (τ, ρ) *startsFrom* φ and, either there is $i \geq 0$ such that $(\tau(i), \rho) \models_{\text{ML}} \varphi'$ or *infinite*(τ). Also, $\Rightarrow_{\mathcal{S}} \models_{\text{RL}} \varphi \Rightarrow \varphi'$ iff $(\tau, \rho) \models_{\text{RL}} \varphi \Rightarrow \varphi'$ for all pairs (τ, ρ) starting from φ with τ complete. The exact definitions are:

```
Definition startsFrom(τ, ρ, φ ⇒ φ') :=∃ γ.τ(1)=γ∧(γ, ρ)⊨ML φ
Definition (τ, ρ) ⊨RL φ ⇒ φ' := (τ, ρ) startsFrom φ ∧
  ((∃ i n γ'. i ≤ n ∧ complete(τ, n) ∧ τ(i) = γ' ∧ (γ', ρ) ⊨ML φ')
                                        ∨ infinite(τ))

Definition ⇒S⊨RL φ ⇒ φ' :=∀ τ ρ. wfPath(τ) ∧ complete(τ) ∧
                        (τ, ρ) startsFrom φ ∧ (τ, ρ) ⊨RL φ ⇒ φ'

Definition ⇒S⊨RL G :=∀ φ ⇒ φ'. φ ⇒ φ' ∈ G→⇒S⊨RL φ ⇒ φ'
```

An ML formula φ is *$\mathcal{S}$-derivable* if there are γ, ρ, and γ' such that $(\gamma, \rho) \models_{\text{ML}} \varphi$ and $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. The $\mathcal{S}$-derivability of an ML formula must not be confused with the notion of *$\mathcal{S}$-derivative* (discussed in Section II-C). To compute the $\mathcal{S}$-derivative of an ML formula in Coq, we use the function:

```
Definition Δ_{φl⇒φr}(φ) := ∃ FV(φl).(φl ∧ φ)^=? ∧ φr
```

Note that, unlike in the definition of derivatives from Section II-C, here we quantify only the variables of φ_l since *wf*$(\varphi_l \Rightarrow \varphi_r)$, that is, $\text{FV}(\varphi_r) \subseteq \text{FV}(\varphi_l)$. We use a similar function for RL formulas:

```
Definition Δ_{φl⇒φr}(φ ⇒ φ') :=∃ FV(φl).(φl ∧ φ)^=?∧φr ⇒ φ',
```

which returns the list of $\mathcal{S}$-derivatives of $\varphi \Rightarrow \varphi'$, and is defined using $\Delta_{\varphi_l \Rightarrow \varphi_r}(\varphi \Rightarrow \varphi')$ for each rule in $\varphi_l \Rightarrow \varphi_r \in \mathcal{S}$.

Next, we introduce two relations $\approx$ and $\sim$, where the former is over RL formulas, and the latter is over pairs of ML formulas and valuations. The relation $\approx$ is used to handle the renaming

of free variables in RL formulas which is required when computing derivatives. More precisely, instead of computing $\Delta_{\varphi_l \Rightarrow \varphi_r}(\varphi \Rightarrow \varphi')$ we compute $\Delta_{\varphi_{l'} \Rightarrow \varphi_{r'}}(\varphi \Rightarrow \varphi')$, where $\varphi_{l'} \Rightarrow \varphi_{r'}$ is $\varphi_l \Rightarrow \varphi_r$ with renamed free variables such that $\mathrm{FV}(\varphi_{l'}) \cap \mathrm{FV}(\varphi) = \emptyset$. This disjointness condition is needed for technical reasons and it is essential for computing derivatives; briefly, if this condition is not met, then there is no guarantee that the derivative above encodes the successors of configurations matching φ. We express the fact that the RL formula $\varphi_{l'} \Rightarrow \varphi_{r'}$ is a renamed version of $\varphi_l \Rightarrow \varphi_r$ using $\approx$: $\varphi_{l'} \Rightarrow \varphi_{r'} \approx \varphi_l \Rightarrow \varphi_r$. If two RL formulas are related by $\approx$ then they denote the same set of transitions between configurations, but with different valuations.

To express the fact that two RL formulas in relation $\approx$ denote the same transitions between configuration, but with different valuations, we use the relation $\sim$:

$$\texttt{Definition}\ (\varphi, \rho) \sim (\varphi', \rho') := \forall\ \gamma. (\gamma, \rho) \models_{\mathrm{ML}} \varphi \leftrightarrow (\gamma, \rho') \models_{\mathrm{ML}} \varphi';$$

Essentially, if $\varphi \sim \varphi'$ then both φ and φ' match over the same set of configurations. Next, we use an axiom to establish the link between relations $\sim$ and $\approx$:

$$\texttt{Axiom}\ \approx^{\sim} : \forall\ \varphi_0\ \varphi_0'\ \varphi_1\ \varphi_1'.\ \varphi_0 \Rightarrow \varphi_0' \approx \varphi_1 \Rightarrow \varphi_1' \rightarrow \forall\ \rho.\ \exists\ \rho'.\ (\varphi_0, \rho) \sim (\varphi_1, \rho') \wedge (\varphi_0', \rho) \sim (\varphi_1', \rho')$$

C. The procedure as a relation

The procedure shown in Figure 1 might not terminate. Because all Coq functions are terminating, the procedure cannot be encoded as a Coq function. Also, the procedure operates with a set of goals instead of a single goal. In our formalisation, we adapt the inference steps above to work with lists of goals, since in Coq it is more convenient to work with lists rather than sets. One could claim that working with lists might restrict the generality of the procedure, but our encoding does not rely on list-specific features. Moreover, in our Coq formalisation we use the relation $\approx$ defined in Section IV-B in order to handle variable renaming explicitly.

In Coq, we introduce non-determinism by encoding the procedure as an inductive relation:

$$\frac{\varphi \Rightarrow \varphi' \in G \qquad M \models_{\mathrm{ML}} \varphi \rightarrow \varphi'}{\mathit{step}(G, G \setminus \{\varphi \Rightarrow \varphi'\})}[\textsc{impl}]$$

$$\frac{\varphi \Rightarrow \varphi' \in G \qquad \begin{array}{ll} \varphi_c \Rightarrow \varphi_c' \in G_0 & \varphi_c \Rightarrow \varphi_c' \approx \varphi_{c'} \Rightarrow \varphi_{c'}' \\ M \models_{\mathrm{ML}} \varphi \rightarrow \overline{\varphi}_c & \mathrm{FV}(\varphi_{c'}) \cap \mathrm{FV}(\varphi) = \emptyset \end{array}}{\mathit{step}(G, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \{\Delta_{\varphi_{c'} \Rightarrow \varphi_{c'}'}(\varphi \Rightarrow \varphi')\})}[\textsc{circ}]$$

$$\frac{\varphi \Rightarrow \varphi' \in G \qquad \varphi\ \mathit{is}\ \mathcal{S}\mathit{-derivable} \qquad \begin{array}{c} \mathcal{S} \approx \mathcal{S}' \\ \mathrm{FV}(\varphi) \cap \mathrm{FV}(\mathcal{S}') = \emptyset \end{array}}{\mathit{step}(G, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \Delta_{\mathcal{S}'}(\varphi \Rightarrow \varphi'))}[\textsc{symb}]$$

Here, $\mathrm{FV}(\mathcal{S}')$ denotes the list of free variables that occur in all the RL formulas in $\mathcal{S}'$. G, G_0, $\mathcal{S}$, and $\mathcal{S}'$ are all lists of RL formulas. By abuse of notation we write $\varphi \Rightarrow \varphi' \in G$ to denote the fact that $\varphi \Rightarrow \varphi'$ is in list G. Intuitively, *step* relates two lists of goals G and G', where G contains the current goal $\varphi \Rightarrow \varphi'$, and G' contains the remaining goals from $G \setminus \{\varphi \Rightarrow \varphi'\}$ and possibly the new goals generated by [CIRC] and [SYMB]. At each step, only a single goal is removed from G, but zero, one, or more goals can be added. Note that, like in the original procedure (Figure 1) which uses sets, it does not matter which goal is chosen from the list G, we only require that $\varphi \Rightarrow \varphi' \in G$. Also, note that derivatives are always computed using equivalent (w.r.t. $\approx$) RL formulas whose variables are conveniently renamed.

A successful execution of the non-deterministic version of the procedure is defined using *steps*:

$$\frac{}{\mathit{steps}(\emptyset)}[\textsc{base}] \qquad \frac{\mathit{step}(G, G') \qquad \mathit{steps}(G')}{\mathit{steps}(G)}[\textsc{steps}]$$

Intuitively, given a set of initial goals G, *step* is applied multiple times until the goals in G and the intermediary ones generated by [CIRC] and [SYMB], are all eliminated by [IMPL].

In [9], for a successful run of the procedure the set $\mathcal{F} \triangleq G_0 \cup \bigcup_i G_i$ is defined, where G_0 is the initial set of goals, $G_1 = \Delta_{\mathcal{S}}(G_0)$, and G_i ($i > 1$) are the sets of goals generated by each recursive call. In Coq, we define the following relation (which says when a goal g is in $\mathcal{F}$):

$$\frac{g \in G_0}{g \in \mathcal{F}}[\textsc{in-}G_0] \qquad \frac{\mathit{step}(G, G') \qquad g \in G \setminus G' \qquad G' \subseteq \mathcal{F}}{g \in \mathcal{F}}[\textsc{in-step}]$$

A goal $g \in \mathcal{F}$ if it is either in G_0, or there is step where g was eliminated (i.e., $\mathit{step}(G, G')$ with $g \in G \setminus G'$) and the remaining goals, including the ones introduced by this step, are also in $\mathcal{F}$. To ensure that this definition is consistent with the one in [9], we prove in Coq the following technical lemma:

Lemma 1: For all lists of goals G, if $\mathit{steps}(G)$ then $G \subseteq \mathcal{F}$.

D. Helper lemmas

In this section we present two helper lemmas that have been also proved in [9], but here we reformulate them more precisely. We also identify and explain the differences w.r.t. [9].

The first lemma states that every goal which is generated by a successful execution is either eliminated by [IMPL], or its left hand side is $\mathcal{S}$-derivable. Unlike in [9], we disentangle the precise hypotheses and formulate it as a stand-alone lemma:

Lemma 2: For all RL formulas $\varphi \Rightarrow \varphi' \in \mathcal{F}$, if $\mathcal{S}$ is total, all goals in G_0 are $\mathcal{S}$-derivable, and there are γ and ρ such that $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$ then $M \models_{\mathrm{ML}} \varphi \rightarrow \varphi'$ or φ is $\mathcal{S}$-derivable.

The main difference between *Lemma* 2 and the corresponding one from [9] is that here we explicitly provide in the hypothesis a configuration γ and a valuation ρ which satisfy φ. The existence of (γ, ρ) is crucial in the soundness proof.

The second lemma states that for every transition $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ which starts from φ there is a symbolic successor φ' of φ such that γ' is an instance of φ'. This lemma is also generalised to paths in Coq, and we prove that for a concrete execution path there is a symbolic one which *covers* it. There are two additional conditions required by this lemma (w.r.t. [9]): all rules from $\mathcal{S}$ have distinct free variables from variables in φ and all formulas from $\mathcal{S}$ and G_0 are well-formed.

Lemma 3: For all transitions $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, for all valuations ρ, and for all formulas φ, if $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$, $f \in \mathcal{S} \cup G_0$ implies $\mathit{wf}(f)$, and $\mathrm{FV}(\mathcal{S}) \cap \mathrm{FV}(\varphi) = \emptyset$, then there are $\alpha \in \mathcal{S}$ and $\varphi' \triangleq \Delta_\alpha(\varphi)$ such that $(\gamma', \rho) \models_{\mathrm{ML}} \varphi'$.

E. The soundness for finite paths

The proof of the soundness theorem depends on an intermediate technical lemma. In this section we formulate the lemma by adding all the hypotheses needed by its proof so that it does not rely on hidden assumptions as in [9]. The lemma can be thought of as a version of soundness for finite paths. We prove it in Coq by induction on the number of transitions in τ.

Lemma 4: For all finite and complete paths τ, for all valuations ρ, and for all formulas $\varphi \Rightarrow \varphi' \in \mathcal{F}$, if (τ, ρ) *startsFrom* φ, $G_0 \neq \emptyset$, *wfPath*(τ), *steps*$(\Delta_{\mathcal{S}}(G_0))$, $\mathcal{S}$ is *total*, the left hand sides of goals in G_0 are $\mathcal{S}$-derivable, all formulas in $\mathcal{S} \cup G_0$ are well formed, and $\mathrm{FV}(\mathcal{S}) \cap \mathrm{FV}(G_0) = \emptyset$, then $(\tau, \rho) \models_{\mathrm{RL}} \varphi \Rightarrow \varphi'$.

With respect to [9], there are several significant differences. First, the proof takes into account the fact that derivatives are computed with renamed RL formulas. Second, we find and fix the flaw from the proof in [9], where a false assumption about the goals in $\mathcal{F}$ is made: given a successful execution of the procedure starting with $\Delta_{\mathcal{S}}(G_0)$, all the goals in $\mathcal{F}$ are eliminated by a step of the procedure. This cannot be true, since $\mathcal{F}$ includes G_0, but the procedure only processes goals starting with $\Delta_{\mathcal{S}}(G_0)$. The assumption does not invalidate the result from [9], but it makes the proof incomplete. In fact, the goals in G_0, which are of interest in the soundness theorem, are not handled. This flaw was detected only when formalising the proof in Coq. To fix it, we add a new hypothesis: the free variables that occur in rules in $\mathcal{S}$ and those that occur in rules in G_0 constitute disjoint sets (i.e., $\mathrm{FV}(\mathcal{S}) \cap \mathrm{FV}(G_0) = \emptyset$).

F. The soundness theorem

In this section we formulate the soundness theorem. Several assumptions from [9] become hypotheses: all formulas in $\mathcal{S}$ and G_0 are well formed and for all the goals $\varphi \Rightarrow \varphi'$ in G_0 φ is $\mathcal{S}$-derivable. Moreover, we add the additional hypothesis required by LEMMA 4: $\mathrm{FV}(\mathcal{S}) \cap \mathrm{FV}(G_0) = \emptyset$. The theorem below is fully formalised and proved in Coq:

Theorem 1: If $\mathcal{S}$ is total, all formulas in $\mathcal{S} \cup G_0$ are well-formed, the left hand sides of goals in G_0 are $\mathcal{S}$-derivable, $\mathrm{FV}(\mathcal{S}) \cap \mathrm{FV}(G_0) = \emptyset$, and *steps*$(\Delta_{\mathcal{S}}(G_0))$ then $\Rightarrow_{\mathcal{S}} \models_{\mathrm{RL}} G_0$.

V. Conclusions

The formal proof of the soundness of the procedure was developed using version `8.4pl5` of the Coq proof assistant. The Coq code[6] is organized as follows: the main file, `sound.v`, contains the definitions for *step* and *steps*, the main lemmas and the soundness theorem, all shown in Section IV. The axioms that we use for ML (Section IV-A) can be found in `ml.v`. The definition of RL, derivatives, and the other related notions from Section IV-B are located in `rl.v` and `derivatives.v`.

The Coq code has over 1600 lines and the proof includes 24 lemmas, 15 axioms (all needed to preserve the language parametric feature) and 1 theorem. It took 5 months for one person to improve the proof in [9] and encode it in Coq.

[6]Zip archive: https://profs.info.uaic.ro/~arusoaie.andrei/soundness-proof.zip

In terms of future work, we are interested in extracting a certified OCaml program from the procedure, and then use it for verification within the $\mathbb{K}$ framework. This requires RL-based semantics of languages in OCaml, which could be extracted from corresponding semantics in Coq.

References

[1] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[2] Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304:57 – 80, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[3] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.

[4] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

[5] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[6] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. *SIGACT News*, 32(1):66–69, 2001.

[7] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.

[8] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[9] Dorel Lucanu, Vlad Rusu, Andrei Arusoaie, and David Nowak. Verifying reachability-logic properties on rewriting-logic specifications. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 451–474. Springer, 2015.

[10] Brandon Moore and Grigore Roşu. Program verification by coinduction. Technical Report http://hdl.handle.net/2142/73177, University of Illinois, February 2015.

[11] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.

[12] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[13] Grigore Roşu. Matching logic — extended abstract. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21, Dagstuhl, Germany, July 2015.

[14] Grigore Roşu and Traian Florin Şerbănuţă. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.

[15] Grigore Roşu and Andrei Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011.

[16] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 555–574. ACM, 2012.

[17] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

[18] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.